CS 4: Lecture 12
Wednesday, March 1, 2006

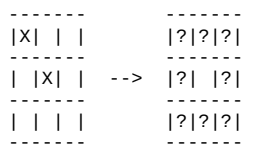
THE GAME OF LIFE

In 1970, the mathematician John Conway invented a "game" played on a grid of square cells--like a chess board, except that it extends infinitely far in all directions.

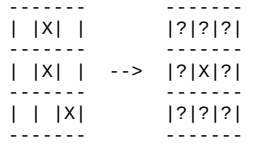
Each cell is in one of two states: it is either _live_ or _dead_. We represent a live cell by putting a marker on the square. Dead cells are empty. Each cell has eight _neighbors_, the other cells that touch it (including the diagonally connected cells).

Life runs one timestep at a time, just like our numerical simulations. After each timestep, each cell's state--live or dead--depends solely on the state of that cell and its neighbors just before the timestep.

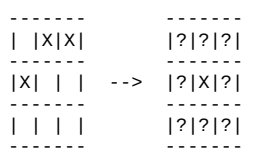
If a live cell has only one or zero live neighbors at the start of a timestep, it dies of loneliness during the timestep. If a live cell has four or more live neighbors at the beginning of a timestep, it dies of overcrowding during the timestep.



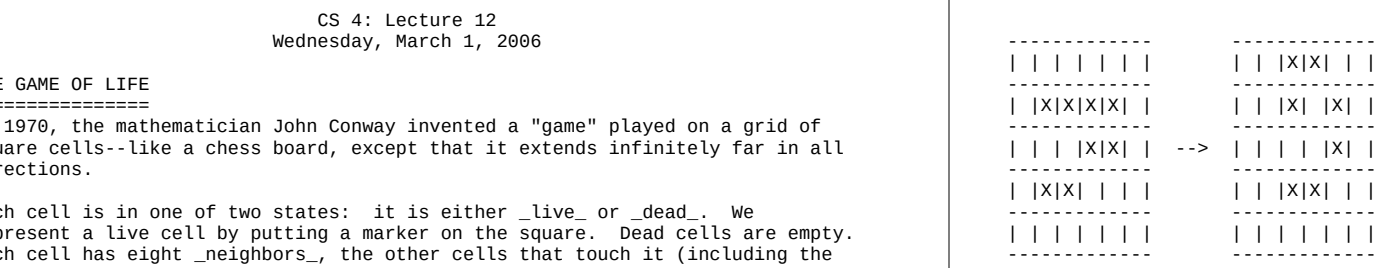
If a live cell has two or three neighbors, it survives the timestep.



If a dead cell has _exactly_ three neighbors, a new life is magically created in the cell during the timestep. Otherwise, it stays dead.



All of the cells change from their old states to their new states simultaneously. The new state of each cell depends _only_ on the states before the timestep took place. You can't decide the new state for one cell based on the new state for its neighbor--if you do that, you'll get it wrong. Here's an example of a timestep.



Those are all the rules. Life is one example of a _cellular_automaton_. You can invent other cellular automata simply by changing the rules--for example, you could decide that a life is also born if seven of its neighbors are live; or you could decide that each cell has one of four different states, with rules as fancy as you like for deciding a cell's state after a timestep. The reason Life is famous, though, is because it exhibits very rich behavior from very simple rules. By choosing your starting state carefully, it's even possible to make the Game of Life simulate a computer.

Cellular automata are sometimes applied to scientific problems. In particular, the Ising model of ferromagnetic materials is effectively a cellular automaton in which each electron has one of two spins. Electrons have a tendency to align their spins with their neighbors, but their spins can also toggle randomly, especially if the material is hot. Cellular automata have also been used to study forest fire propagation and the percolation of fluids through porous materials.

Implementing Life

How could you implement the Game of Life? Well, a computer can't truly represent an infinite game board, so we're going to compromise a bit and use a finite game board. For example, we might create the game board like this.

```
int[][] board = new int[150][50];
```

Each array element will be zero if the corresponding cell is dead, and one if the cell is live. We use special names to define these constants.

```
public class Automaton {
  private final static int DEAD = 0;
  private final static int LIVE = 1;
```

Java's "final" keyword declares a value that can never be changed. Because DEAD is "final", it's illegal to assign anything to it (except in the initializer). If you try, you'll get a compiler error.

We could have used booleans instead of ints, but we'd like our code to generalize to more complicated automata, where each cell might have more than two states.

Life isn't interesting if we start with all the cells dead, so we need to come up with an interesting starting state. Recall from lab that Java allows you to construct a Random object, which you can use to generate random numbers.

```
    Random r = new Random(new Date().getTime());
    final double LIVING_PROBABILITY = 0.40;

    // initialize the array
    for (int i = 0; i < board.length; i++) {
      for (int j = 0; j < board[i].length; j++) {
        if (r.nextDouble() < LIVING_PROBABILITY) {
          board[i][j] = LIVE;
        } else {
          board[i][j] = DEAD;
        }
      }
    }
```

What's going on here?

- A Random object generates random numbers for us. However, a random number generator needs a "seed" number to get it started. If it always starts with the same seed, you'll always get the same random numbers, so you'll watch the same Game of Life each time you run it, and that's not very interesting. So we use the time of day to initialize the random number generator when we construct the Random object. That way, we get a different game every time.
- We initialize each cell to be live with 40% probability, and dead with 60% probability.

Next, we implement a "step" method that performs one timestep.

```
        public static int[][] step (int[][] in) {
```

Observe that step takes in a two-dimensional array, and also returns one. The array it returns is _not_ the same array object as the input parameter. The "step" method constructs a new array, writes the updated state in it, and returns the new array. Why a _new_ array? Suppose we tried to write the new state in the same array as the old state. To determine the new state of a cell, you need to know the _old_ state of all its neighbors. But if you update a cell in the same array, then you erase its old state--so you can't update its neighbors. This is a common programming error.

You'll write the "step" method in lab. You'll need to figure out how to find a cell and its neighbors, determine their states, and figure out the cell's new state.

CONSTANTS
=========
If you find yourself repeatedly using a numerical value with some "meaning" in your code, you should probably turn it into a "final" constant.

```
BAD:    if (month == 2) {

GOOD:   public final static int FEBRUARY = 2;    // Usually near top of class.
        ...
        if (month == FEBRUARY) {
```

Why? Because if you ever need to change the numerical value assigned to February, you'll only have to change one line of code, rather than hundreds.

The custom of rendering constants in all-caps is long-established and was inherited from C. The "final" keyword ensures that a programmer can't accidentally write code that changes the value during program execution.

For any array x, "x.length" is a "final" field.